Logit Virtual Machines: Deterministic Database Execution over Neural Natural Parameters

Jaepil Jeong
Cognica, Inc.
Email: jaepil@cognica.io
Date: May 6, 2026

"What can be shown, cannot be said."
— Ludwig Wittgenstein

Abstract

Neural systems and databases are usually connected by an external integration layer: the model emits a query-like text or vector, the database retrieves records, and application code converts the result into a prompt, embedding, or auxiliary context. This architecture treats the database as a tool outside the model rather than as a computational substrate inside inference. We propose **Logit Virtual Machines** (LVMs), a deterministic execution model in which a database acts as a virtual machine over logit-valued relations. A neural model emits logits over instructions, schema objects, fields, terms, values, candidate handles, registers, control decisions, and output symbols. A trainable typed compiler maps these logits to executable database programs while enforcing grammar and schema constraints. The database executes the resulting programs over relational, textual, vector, and graph indexes, and returns logit deltas in the model's output space through a typed `ACCUM` projection.

The central claim is that logits are not merely decoding scores; they are executable natural parameters. Query execution can therefore be formulated as deterministic transformation of logit states rather than stochastic retrieval or external tool invocation. We define logit-valued relations over log semirings, construct a typed instruction algebra for database execution, give a small-step operational semantics for the LVM, solve the compiler as a type-safe neural program synthesizer, solve the training problem through latent deterministic execution, and prove determinism, closure, support preservation, compiler type safety, compiler realizability, compositionality, differentiable training without differentiating through the database, zero-temperature convergence to deterministic execution, and equivalence between logit addition and normalized Product-of-Experts aggregation. Classical Boolean query algebra is recovered as the support projection of logit-valued execution, while ranked retrieval, vector calibration, graph traversal, and transactional memory writes arise as weighted extensions of the same execution model.

This framework does not pretend that model-database alignment is magically absent. The alignment map is made explicit as a typed, trainable, verifiable component of the VM semantics. What disappears is not every projection, but the untyped application-specific glue layer whose behavior is outside the formal model. Physical database indexes remain necessary as evaluators of the instruction set, just as matrix kernels evaluate linear algebra; however, the interface between neural inference and database execution becomes a typed logit transition rather than an application-specific procedure. The result is a deterministic, durable, index-aware memory model for neural computation.

---

1. Introduction

1.1 The Integration Problem

Modern neural systems increasingly rely on external memory. In practice, this memory is usually implemented by a database, search index, vector store, graph store, or hybrid data engine. The dominant integration pattern is procedural:

1. A neural model emits text, embeddings, or tool-call arguments.
2. Application code translates the model output into a database query.
3. The database returns records or text chunks.
4. Application code serializes the result back into a prompt or feature vector.
5. The neural model continues inference.

This architecture works, but it is not a unified computational model. The model and the database are separated by a semantic glue layer whose behavior is neither a neural operation nor a database algebraic operation. It is code. The goal of this paper is not to deny that an alignment map exists; it is to make that map typed, trainable, deterministic at inference, and subject to the same formal semantics as the rest of the system.

The problem is not that code exists. Every physical system requires code. The problem is that the **meaning** of the integration is located in application code rather than in a mathematical interface. Matrix multiplication is implemented by kernels, but its semantics are linear algebra. Database-neural integration should be similar: physical indexes should evaluate a mathematical instruction set, not define the semantics of integration.

1.2 From Retrieval to Execution

Retrieval treats the database as a source of auxiliary context. Execution treats the database as a virtual machine. The distinction is fundamental.

In the retrieval view, the model asks:

$$\text{Which records should be returned?} \tag{1}$$

In the execution view, the model provides a state, and the database computes:

$$\text{What is the next logit state?} \tag{2}$$

The output of the database is not a chunk of text but a transformation:

$$\Delta\ell \in \mathbb{R}^{Y}, \tag{5}$$

where Y is an output domain such as tokens, entities, documents, actions, or fields. The final neural state is obtained by deterministic logit addition:

$$\ell_{\text{final}} = \ell_{\text{model}} + \Delta\ell_{\text{db}}. \tag{6}$$

Sampling, if used, occurs after this deterministic computation. The virtual machine itself is deterministic.

1.3 Main Thesis

The thesis of this paper is:

> A database can be interpreted as a deterministic virtual machine over logit-valued relations. Neural logits are executable natural parameters. Query execution is deterministic transformation of logit states, and database retrieval is a VM instruction rather than an external tool call.

This thesis separates two concepts often conflated:

- **Probabilistic semantics:** logits may be interpreted as natural parameters of probability distributions.
- **Deterministic execution:** the computation transforming logits into new logits is a deterministic transition.

Neural forward passes already have this structure. A neural network deterministically maps input tensors to logits; the logits may later parameterize a probability distribution. We extend the same principle to database execution.

1.4 Contributions

This paper makes the following contributions.

1. **Logit-valued relations.** We define relations whose tuple weights live in a log semiring. Boolean relations and posting lists are recovered as the special case where weights are 0 for present tuples and $-\infty$ for absent tuples.
2. **A deterministic database VM.** We define a typed virtual machine whose registers contain logit-valued relations, schema symbols, scalar logits, and transaction buffers.
3. **Instruction algebra.** We formalize database operations such as seek, filter, union, intersection, join, projection, graph traversal, scoring, accumulation, write, and commit as deterministic state transitions.
4. **Neural-to-database compilation.** We define compilation as a first-class part of the LVM: a learned scorer over typed grammar derivations followed by deterministic lowering into VM instructions.
5. **Compiler construction and learning.** We prove compiler totality for finite typed grammars with deterministic tie-breaking, define self-supervised and inverse-execution trace generation, and show that annealed soft compilation converges to the deterministic inference-time compiler.
6. **Closure and support preservation.** We prove that LVM instructions are closed over logit-valued relations and that classical Boolean query algebra is recovered by support projection.

7. **Logit aggregation theorem.** We prove that adding database-produced logit deltas to model logits is equivalent, after softmax, to normalized multiplicative expert composition.
8. **Semantic glue localization.** We show that the unavoidable entity-to-output alignment map is not external glue: it is the typed `ACCUM` projection, which may be fixed, learned, or hybrid, trained jointly with the compiler, and verified by the same type system.
9. **Trainability without differentiating through the database.** We formulate training as latent deterministic execution over a bounded typed program space. The database remains a non-differentiable deterministic evaluator, while gradients flow through neural program scores, compiler derivation scores, base logits, and learned output projections.
10. **Transactional memory semantics.** We define durable memory writes as deterministic database state transitions rather than differentiable matrix writes.

---

2. Mathematical Preliminaries

2.1 Logits

Let X be a finite domain. A **logit vector** over X is a function

$$\ell : X \to \mathbb{R} \cup \{-\infty\}. \tag{7}$$

The value $\ell(x)$ is interpreted as unnormalized evidence for x. The value $-\infty$ means that x is impossible or absent.

When a probability interpretation is desired, the normalized distribution is

$$\mathrm{softmax}(\ell)(x) = \frac{\exp(\ell(x))}{\sum_{u \in X} \exp(\ell(u))}. \tag{8}$$

The softmax is not part of the VM execution semantics. It is an optional interpretation layer.

2.2 The Log Semiring

Let

$$K_{\log} = \mathbb{R} \cup \{-\infty\}. \tag{9}$$

We define two useful semiring structures on $K_{\log}$.

Definition 2.2.1 (Log-sum-exp semiring). The log-sum-exp semiring is

$$(K_{\log}, \oplus, \otimes, -\infty, 0), \tag{10}$$

where

$$a \oplus b = \log(\exp(a) + \exp(b)), \tag{11}$$

and

$$a \otimes b = a + b. \tag{12}$$

The additive identity is $-\infty$ and the multiplicative identity is 0.

Definition 2.2.2 (Max-plus semiring). The max-plus semiring is

$$(K_{\log}, \oplus_{\max}, \otimes, -\infty, 0), \tag{13}$$

where

$$a \oplus_{\max} b = \max(a, b), \tag{14}$$

and

$$a \otimes b = a + b. \tag{15}$$

The log-sum-exp semiring accumulates all evidence paths. The max-plus semiring keeps only the strongest path. Both are deterministic.

Remark 2.2.3 (Choice of semiring). Unless otherwise stated, K denotes either $K_{\log}$ with log-sum-exp addition or $K_{\log}$ with max-plus addition. Theorems about closure and determinism hold for both. The support-preservation theorems require only that $-\infty$ is absorbing under $\otimes$ and that $a \oplus b > -\infty$ whenever at least one of a, b is finite.

2.3 Logit-Valued Relations

Let $A_1, \dots, A_n$ be finite domains. A classical relation over these domains is a subset of

$$A_1 \times \cdots \times A_n. \tag{16}$$

A **logit-valued relation** is a function

$$R : A_1 \times \cdots \times A_n \to K. \tag{17}$$

The value $R(a_1, \dots, a_n)$ is the logit weight of the tuple.

Definition 2.3.1 (Support). The support of a logit-valued relation R is

$$\operatorname{supp}(R) = \{x : R(x) > -\infty\}. \tag{18}$$

Definition 2.3.2 (Boolean embedding). Given a classical relation $B \subseteq A_1 \times \cdots \times A_n$, its Boolean embedding is the logit-valued relation $\eta(B)$ defined by

$$\eta(B)(x) = \begin{cases} 0, & x \in B, \\ -\infty, & x \notin B. \end{cases} \tag{19}$$

Thus classical relations are embedded as zero-weight supports inside logit-valued relations.

2.4 Posting Lists as Sparse Relations

Let $\mathcal{D}$ be a finite document or entity domain. A posting list is a sparse representation of a relation

$$L : \mathcal{D} \to K. \tag{21}$$

In the Boolean case, $L(d) = 0$ means document d is present, and $L(d) = -\infty$ means it is absent. In the weighted case, $L(d)$ is the logit evidence associated with d.

Remark 2.4.1 (Unification). This definition covers inverted indexes, vector-search candidate lists, relational filters, graph traversal frontiers, and ranked result sets. They differ physically, but semantically they are logit-valued sparse relations.

2.5 Structural Absence vs. Evidence Neutrality

Two different notions of absence must be kept separate.

A relation uses semiring absence:

$$R(x) = -\infty \tag{22}$$

meaning that tuple x is not in the support of the relation. This is structural absence. It is appropriate for posting lists, joins, filters, graph frontiers, and all support-level database operations.

An output accumulator uses evidence neutrality:

$$\omega(y) = 0 \tag{23}$$

meaning that the database contributes no evidence for or against output symbol y. This is the multiplicative identity under Product-of-Experts logit fusion, because

$$\ell_{\text{model}}(y) + 0 = \ell_{\text{model}}(y). \tag{24}$$

Therefore the VM uses two different identities:

Object	Domain	Absence / identity	Meaning
Relation tuple	$R : X \to K$	$-\infty$	tuple is absent from support
Output evidence	$\omega : Y \to \mathbb{R}$	0	no database evidence; model logit unchanged

Principle 2.5.1 (Neutral outside support). Relation support controls which tuples participate in execution. Output evidence is neutral outside the projected support. A tuple absent from a relation contributes nothing to the output accumulator; it does not contribute $-\infty$ to the final model logit.

This distinction prevents a common error: Boolean embedding uses $-\infty$ for absent tuples, but `ACCUM` must use 0 for absent output evidence. Otherwise a missing database result would erase the model's own distribution rather than leaving it unchanged.

---

3. The Logit Database Virtual Machine

3.1 Database State

A database memory state is a tuple

$$\mathcal{M} = (\Sigma, \mathcal{D}, \mathcal{I}, \mathcal{G}, \mathcal{T}, \tau), \tag{25}$$

where:

- Σ is a schema containing entity types, fields, relation symbols, graph labels, and value domains.
- $\mathcal{D}$ is the durable set of documents, tuples, entities, or records.
- $\mathcal{I}$ is a family of indexes, including text, vector, relational, and graph indexes.
- $\mathcal{G}$ is an optional graph structure over entities.
- $\mathcal{T}$ is a collection of index statistics, calibration parameters, and cost bounds.
- τ is a transaction log or version state.

The VM may read from $\mathcal{M}$ and may update $\mathcal{M}$ through transactional instructions.

3.2 Register Space

The VM has typed registers. Let

$$\mathcal{R} = \mathcal{R}_{\text{rel}} \cup \mathcal{R}_{\text{sym}} \cup \mathcal{R}_{\text{logit}} \cup \mathcal{R}_{\text{buf}}. \tag{26}$$

The register classes are:

- Relation registers: hold logit-valued relations.
- Symbol registers: hold schema objects, terms, field names, graph labels, or entity IDs.
- Logit registers: hold scalar or vector logits.
- Buffer registers: hold pending writes before commit.

A register valuation is a partial typed map

$$\rho : \mathcal{R} \to_{\text{partial}} \mathcal{V}, \tag{27}$$

where $\mathcal{V}$ is the union of all typed VM values.

3.3 VM State

A VM state is

$$s = (pc, \rho, \mathcal{M}, \omega), \tag{28}$$

where:

- pc is the program counter.
- ρ is the register valuation.

- $\mathcal{M}$ is the database memory state.
- ω is the output evidence accumulator, typically a logit delta vector over tokens, entities, documents, or actions. It is initialized as $\omega_0(y) = 0$ for every output symbol y.

The initial state is denoted s_0.

3.4 Instruction Set

An instruction is a typed partial function on states. The core instruction classes are:

```
LOAD_SYMBOL     neural-logits -> symbol register
SEEK_TERM       term register -> relation register
SEEK_VECTOR     vector register -> relation register
FILTER          relation register, predicate -> relation register
UNION           relation register, relation register -> relation register
INTERSECT       relation register, relation register -> relation register
DIFFERENCE      relation register, relation register -> relation register
JOIN            relation register, relation register, key -> relation register
PROJECT         relation register, attributes -> relation register
TRAVERSE        graph frontier, edge label, hop bound -> relation register
SCORE_TEXT      relation register, query state -> relation register
SCORE_VECTOR    relation register, query vector -> relation register
SCORE_GRAPH     relation register, graph pattern -> relation register
ACCUM           relation register, typed evidence projection -> output logits
WRITE           typed delta -> transaction buffer
COMMIT          transaction buffer -> durable database state
HALT            stop execution
```

The instruction set is intentionally database-like. It is not a neural layer inserted into the model. It is a typed algebra of database execution whose inputs and outputs are logits.

3.5 Small-Step Semantics

Let $\mathcal{A}$ be the set of well-typed instructions. The small-step transition function is

$$\delta : \mathcal{S} \times \mathcal{A} \to \mathcal{S}, \tag{29}$$

where $\mathcal{S}$ is the set of VM states.

For a program

$$P = (a_0, a_1, \dots, a_{n-1}), \tag{30}$$

the execution sequence is

$$s_{t+1} = \delta(s_t, a_{pc_t}). \tag{31}$$

Execution halts when the current instruction is `HALT` or when $pc = n$.

Definition 3.5.1 (Program evaluation). The result of running program P on initial state s_0 and database state $\mathcal{M}$ is

$$\mathrm{Eval}_{\mathcal{M}}(P, s_0) = s_N, \tag{32}$$

where s_N is the unique halted state if execution terminates.

---

4. Instruction Semantics over Logit-Valued Relations

This section defines representative instructions. Let R, S be logit-valued relations over suitable domains.

4.1 Union

For relations $R, S : X \to K$, define

$$(R \cup_K S)(x) = R(x) \oplus S(x). \tag{33}$$

This accumulates alternative evidence for the same tuple.

4.2 Intersection

For relations $R, S : X \to K$, define

$$(R \cap_K S)(x) = R(x) \otimes S(x) = R(x) + S(x). \tag{34}$$

This accumulates conjunctive evidence: the tuple must be supported by both inputs.

4.3 Difference

The cleanest deterministic difference operator acts on support:

$$(R \setminus_K S)(x) = \begin{cases} R(x), & S(x) = -\infty, \\ -\infty, & S(x) > -\infty. \end{cases} \tag{35}$$

This is a support-level exclusion. More refined anti-evidence versions can be defined, but support difference is the canonical database operation.

4.4 Filter

Let $\varphi : X \to \{0, 1\}$ be a deterministic predicate. Define

$$\mathrm{Filter}_\varphi(R)(x) = \begin{cases} R(x), & \varphi(x) = 1, \\ -\infty, & \varphi(x) = 0. \end{cases} \tag{36}$$

Filters are therefore multiplicative masks in the log semiring.

4.5 Join

Let

$$R : X \to K, \tag{37}$$

and

$$S : Y \to K. \tag{38}$$

Let $\theta : X \times Y \to \{0, 1\}$ be a deterministic join predicate. The logit-valued join is

$$(R \bowtie_\theta S)(x, y) = \begin{cases} R(x) \otimes S(y), & \theta(x, y) = 1, \\ -\infty, & \theta(x, y) = 0. \end{cases} \tag{39}$$

When $\otimes$ is log addition, the joined tuple's evidence is the sum of the two input evidences.

4.6 Projection

Let $\pi : X \to Y$ be a deterministic projection map. The projected relation is

$$\mathrm{Project}_\pi(R)(y) = \bigoplus_{x \in X : \pi(x) = y} R(x). \tag{40}$$

Projection aggregates all tuples mapping to the same output tuple. In the log-sum-exp semiring, projection sums evidence over all paths. In the max-plus semiring, projection selects the strongest path.

4.7 Graph Traversal

Let $E : V \times V \times L_G \to K$ be a logit-valued edge relation over vertices V and labels L_G. Let $F : V \to K$ be a frontier relation. For label $r \in L_G$, one-hop traversal is

$$\mathrm{Traverse}_r(F)(v') = \bigoplus_{v \in V} F(v) \otimes E(v, v', r). \tag{41}$$

The k-hop traversal is repeated composition:

$$\mathrm{Traverse}_r^k(F) = (\mathrm{Traverse}_r)^k(F). \tag{42}$$

This is graph search as semiring matrix multiplication over adjacency relations.

4.8 Scoring Instructions

A scoring instruction maps a candidate relation into another relation by adding deterministic evidence from an index or statistic.

Let $R : \mathcal{D} \to K$ be a candidate relation and let

$$e : \mathcal{D} \to K \tag{43}$$

be an index-derived evidence function. Then

$$\mathrm{Score}_e(R)(d) = R(d) \otimes e(d) = R(d) + e(d). \tag{44}$$

The evidence function may come from text statistics, vector density ratios, graph centrality, recency, authority, access control, or learned calibration. The VM requires only that $e(d)$ be a deterministic function of the database state, query state, and index statistics.

4.9 Accumulation into Output Logits

`ACCUM` is the boundary between database tuples and model-output logits. This boundary is not trivial, especially when relation tuples are entities or records and the output domain is an open vocabulary. We therefore define it explicitly.

Let $R : X \to K$ be a relation register, where X may be documents, entities, joined tuples, graph vertices, field values, or transaction deltas. Let Y be the output domain. A projection kernel is a typed map

$$\psi : X \times Y \to K. \tag{45}$$

For a given output symbol y, define the contributing support

$$C_y(R, \psi) = \{x \in \mathrm{supp}(R) : \psi(x, y) > -\infty\}. \tag{46}$$

The projected contribution is

$$\chi_{R,\psi}(y) = \begin{cases} \bigoplus_{x \in C_y(R,\psi)} R(x) \otimes \psi(x, y), & C_y(R, \psi) \neq \emptyset, \\ 0, & C_y(R, \psi) = \emptyset. \end{cases} \tag{47}$$

The `ACCUM` instruction updates the output accumulator by ordinary logit addition:

$$\omega'(y) = \omega(y) + \chi_{R,\psi}(y). \tag{48}$$

The use of 0 in the empty-support case is essential. It means that if no database tuple projects to output y, then the database contributes no evidence about y and the model logit remains unchanged under fusion.

4.10 Forms of Projection Kernels

The projection kernel ψ can take three forms.

Canonical projection. If X and Y share identifiers, or if the database stores canonical output symbols, then ψ is a deterministic lookup:

$$\psi(x, y) = \begin{cases} 0, & y = \mathrm{canonical}(x), \\ -\infty, & \mathrm{otherwise}. \end{cases} \tag{49}$$

This covers entity IDs, foreign keys, schema values, enumerations, and canonical names.

Lexicalized projection. If an entity has aliases, descriptions, or surface forms, ψ is a typed finite lexicalizer:

$$\psi(x, y) = \lambda_{\mathrm{alias}}(x, y), \tag{50}$$

where λ_{alias} is a deterministic table or index-derived score over known aliases and tokenizations.

Learned projection. If no canonical mapping exists, ψ is parameterized:

$$\psi_\phi(x, y). \tag{51}$$

The parameters ϕ are trained from execution traces and output likelihood. The learned projection is still inside the VM type system: its domain, range, calibration constraints, and evidence budget are explicit.

Principle 4.10.1 (Projection boundary). The LVM does not make the tuple-to-output problem disappear. It turns that problem into a typed projection kernel that can be fixed, lexicalized, learned, calibrated, audited, and tested. The contribution is not the absence of projection; it is the elimination of untyped projection code.

---

5. Typed Neural Compilation

The compiler is a first-class component of the LVM. It is not an external integration layer and not a deferred implementation detail. The compiler is the typed neural mechanism that converts model logits into verified VM programs.

The design goal is:

$$\text{neural logits} \quad \longrightarrow \quad \text{well-typed deterministic program.} \tag{52}$$

This section defines the compiler. Section 13 gives the training theory.

5.1 Sparse Interactive Logit Interface

A literal vector

$$\ell^{\text{entity}} \in \mathbb{R}^{|\mathcal{D}|} \tag{53}$$

is not a scalable interface when $|\mathcal{D}|$ is large. In realistic databases, $|\mathcal{D}|$ may be 10^9 or more. The LVM therefore does not require flat materialization of logits over the global entity universe.

At compiler step i, the model and VM expose a **materialized typed domain**

$$C_i(\tau) \subseteq \mathrm{Dom}(\tau), \tag{54}$$

where τ is a type such as opcode, field, term, value, register, entity handle, document handle, graph vertex, or transaction symbol. The compiler emits logits only over the currently materialized finite set:

$$\ell_i^\tau : C_i(\tau) \to \mathbb{R}. \tag{55}$$

The candidate set $C_i(\tau)$ is produced by the grammar, schema, current registers, previous instructions, and database index calls. For example:

- opcode candidates come from the grammar state;
- field candidates come from the current schema type;
- term candidates come from lexical analyzers or finite prefix tables;
- entity candidates come from a prior `SEEK`, `FILTER`, `JOIN`, `TRAVERSE`, or `TOPK` instruction;
- register candidates come from the current VM register valuation.

Thus the entity interface is a sparse pointer interface, not a flat softmax over all entities.

The structured logit state at compiler step i is therefore

$$\ell_i = \{\ell_i^\tau : \tau \in \mathrm{Types}(q_i, \rho_i)\}, \tag{56}$$

where q_i is the compiler state and ρ_i is the current VM register valuation.

Definition 5.1.1 (Materialization operator). For each type τ, a materialization operator

$$\mathrm{Mat}_\tau(q_i, \rho_i, \mathcal{M}, x) \to C_i(\tau) \tag{57}$$

returns the finite candidate domain visible to the compiler at step i.

Definition 5.1.2 (Pointer logit). A pointer logit over type τ is a logit vector over $C_i(\tau)$ rather than over $\mathrm{Dom}(\tau)$:

$$\ell_i^\tau : C_i(\tau) \to \mathbb{R}. \tag{58}$$

The selected item is a durable database handle, not an embedded copy of the entity.

Theorem 5.1.3 (Finite materialization). If every materialization operator returns a finite candidate set and the grammar has finite branching over non-materialized symbols, then each compiler step has finite branching, regardless of the cardinality of the global entity domain.

Proof. At step i, valid fragments are drawn either from a finite grammar branch or from some $C_i(\tau)$. By assumption, every $C_i(\tau)$ is finite. A finite union of finite sets is finite. □

5.1.4 Chain-of-Instruction Protocol

Compilation is interactive. The compiler does not need to decode an entire database program in one pass before touching the database. It alternates between typed instruction emission and deterministic VM feedback:

$$(q_i, \rho_i, \mathcal{M}) \xrightarrow{\text{Mat}} C_i \xrightarrow{\text{score}} a_i \xrightarrow{\delta} (q_{i+1}, \rho_{i+1}, \mathcal{M}). \tag{59}$$

The next candidate domain depends on the previous instruction's result. A typical sequence is:

```
choose SEEK_TERM over finite term candidates
execute SEEK_TERM -> materialize posting-list register r1
choose FILTER fields valid for r1's entity type
execute FILTER -> materialize smaller relation r2
choose PROJECT or JOIN over fields valid for r2
execute PROJECT/JOIN -> materialize candidate handles
choose ACCUM projection over the resulting handle type
```

This is a turn-taking protocol between compiler and database. The compiler is therefore closer to a typed controller or pointer-network over database state than to a single-pass autoregressive text decoder.

Remark 5.1.5 (No global entity softmax). Symbols such as ℓ^{entity} always mean pointer logits over a materialized candidate set unless explicitly stated otherwise. The VM never requires a dense vector of length $|\mathcal{D}|$.

5.2 Typed Instruction Grammar

Let Γ be a typing context containing schema types, register types, field domains, instruction signatures, and transaction permissions.

A typed instruction grammar is a finite-state system relative to active views:

$$\mathcal{G}_\Gamma = (Q, A_{\text{schema}}, E, q_0, F), \tag{60}$$

where:

- Q is a finite set of compiler states,
- A_{schema} is a finite set of instruction-fragment schemas,
- dynamic argument fragments are drawn from finite materialized candidate sets $C_i(\tau)$,
- E is the transition relation over states and instantiated fragments,
- q_0 is the initial compiler state,
- $F \subseteq Q$ is the set of accepting states.

A transition

$$(q, a, q') \in E \tag{61}$$

means that fragment a is well-typed in compiler state q and updates the typing context to state q'.

Examples of fragments include opcodes, argument choices, register references, field names, constants, and `HALT`.

A complete program is a path

$$q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{n-1}} q_n, \tag{62}$$

with $q_n \in F$. The emitted program is

$$P = (a_0, \ldots, a_{n-1}). \tag{63}$$

5.3 Compiler Scoring Function

The neural compiler assigns scores to valid next fragments using the input, grammar state, instruction history, current registers, and materialized candidate domains:

$$g_\theta(a \mid x, q, a_{<i}, \rho_i, C_i) \in \mathbb{R}. \tag{64}$$

Invalid fragments are masked by the grammar:

$$M_\Gamma(a \mid q) = \begin{cases} 0, & \exists q' : (q, a, q') \in E, \\ -\infty, & \text{otherwise.} \end{cases} \tag{65}$$

The masked score is

$$\tilde{g}_\theta(a \mid x, q, a_{<i}, \rho_i, C_i) = g_\theta(a \mid x, q, a_{<i}, \rho_i, C_i) + M_\Gamma(a \mid q). \tag{66}$$

5.4 Stable Deterministic Decoding

Fix a total order $<$ over all instruction fragments. At state q_i, the compiler chooses

$$a_i = \operatorname{argmax}_< \tilde{g}_\theta(a \mid x, q_i, a_{<i}, \rho_i, C_i). \tag{67}$$

The next state q_{i+1} is the unique state determined by the grammar transition selected by a_i. If multiple target states are possible, the grammar refines the fragment with an explicit deterministic transition ID, so the transition relation becomes functional.

The compiler halts when it emits `HALT` in an accepting state or reaches a fixed maximum program length T.

Thus the deterministic compiler is

$$\kappa_{\theta,\Gamma,T}(x, s_0^{\text{vm}}) = (a_0, \dots, a_{n-1}). \tag{68}$$

5.5 Verified Compiler Theorem

Theorem 5.5.1 (Compiler validity). For any input x, initial VM state s_0^{vm}, typing context Γ, and maximum length T, the deterministic compiler $\kappa_{\theta,\Gamma,T}$ emits either a well-typed program in $\mathcal{P}_{\Gamma,T}$ or a distinguished well-typed failure program `NOOP HALT`.

Proof. At each step, invalid fragments receive score $-\infty$ and cannot be selected by stable argmax. Therefore every emitted fragment corresponds to a valid transition in $\mathcal{G}_\Gamma$. The emitted sequence is a path in the grammar. If the path reaches an accepting state, it is a well-typed program. If no accepting state is reached by length T, the machine emits the distinguished `NOOP HALT`, which is included in the grammar as a well-typed program. □

Corollary 5.5.2 (No ill-typed database calls). A compiled LVM program cannot reference a field, register, relation, graph label, or transaction operation outside the typing context. Type safety is enforced before database execution.

5.6 Compiler Realizability

Let $\mathcal{P}_{\Gamma,T}$ be the finite set of well-typed programs of length at most T.

Theorem 5.6.1 (Realizability of any bounded typed program). For any target program $P^* \in \mathcal{P}_{\Gamma,T}$, there exists a compiler scoring function g such that

$$\kappa_{g,\Gamma,T}(x, s_0^{\text{vm}}) = P^* \tag{69}$$

for every input x for which P^* is the desired program.

Proof. Let $P^* = (a_0^*, \dots, a_{n-1}^*)$. Define the score at each step by

$$g(a_i^* \mid x, q_i, a_{<i}^*, \rho_i, C_i) = 1 \tag{70}$$

and

$$g(a \mid x, q_i, a_{<i}^*, \rho_i, C_i) = 0 \tag{71}$$

for every other valid fragment a. Invalid fragments are masked to $-\infty$. Stable argmax therefore selects a_i^* at each step. After the final fragment, assign highest score to `HALT`. Thus the emitted program is exactly P^*. □

Remark 5.6.2 (Meaning of realizability). This theorem is not a claim that a finite neural network automatically learns every program. It states that the compiler formalism itself does not block any bounded well-typed program. Learning is addressed in Section 13.

5.7 Cost-Aware Compilation

A compiler may include a deterministic cost model

$$C : \mathcal{P}_{\Gamma,T} \to \mathbb{R}_{\geq 0}. \tag{72}$$

The program score becomes

$$S_\theta^{\text{cost}}(P \mid x) = S_\theta(P \mid x) - \lambda_C C(P). \tag{73}$$

This makes physical plan cost part of the compiler objective while preserving semantic determinism. The chosen program is still the stable argmax of a typed score.

5.8 Deterministic Beam Compiler

For training and high-recall inference, the compiler can return a finite program relation rather than a single program. With beam width B, the compiler returns

$$\mathcal{B}_\theta(x) = \text{TopB}_<\{S_\theta(P \mid x) : P \in \mathcal{P}_{\Gamma,T}\}. \tag{74}$$

The beam is deterministic when the grammar, scores, beam width, and tie-breaking order are fixed. It is represented as a logit-valued relation over programs:

$$R_{\text{prog}}(P) = \begin{cases} S_\theta(P \mid x), & P \in \mathcal{B}_\theta(x), \\ -\infty, & \text{otherwise.} \end{cases} \tag{75}$$

This relation is the bridge between deterministic inference and trainable latent execution.

5.9 Compiler Summary

The compiler is solved at four levels:

1. **Syntactic validity:** grammar masks and type states prevent ill-typed programs.
2. **Scalable addressing:** logits over large domains are pointer logits over materialized candidate sets, not dense vectors over the database universe.
3. **Deterministic execution:** stable argmax and fixed tie-breaking produce a unique instruction at inference, with database feedback materializing the next candidate set.
4. **Trainability:** program scores are learned by the trace and latent execution objectives of Section 13.

The compiler is therefore part of the LVM semantics, not an external unsolved layer.

---

6. The Complete Inference Cycle

The LVM inference cycle consists of deterministic maps, but these maps are not invoked at every autoregressive token by default. Database execution occurs at typed trigger points.

6.0 Triggered Invocation Semantics

Let t index autoregressive decoding steps. A trigger policy is a deterministic function

$$\chi_t = \text{Trigger}_\theta(x_{\leq t}, s_t) \in \{0, 1\}, \tag{76}$$

optionally constrained by type rules, latency budgets, cache validity, or task state. The compiler is invoked only when $\chi_t = 1$.

A trigger may correspond to:

- an explicit entity, citation, calculation, update, or lookup need;
- a schema-constrained slot in a structured generation task;
- a low-confidence region of the neural decoder;

- a boundary between planning and surface realization;
- a coarse query-level, paragraph-level, or transaction-level retrieval event.

An invocation produces an evidence episode

$$E_e = (\omega_e, \nu_e), \tag{77}$$

where $\omega_e : Y \to \mathbb{R}$ is the database evidence map and ν_e is a deterministic validity predicate for reusing that evidence. At token t, the active VM contribution is

$$\omega_t^{\mathrm{db}} = \begin{cases} \omega_e, & \text{some cached episode } E_e \text{ is valid at } t, \\ 0, & \text{no valid cached episode exists and } \chi_t = 0. \end{cases} \tag{78}$$

If $\chi_t = 1$, a new episode is executed and may replace or extend the cache. Therefore the LVM supports per-query, per-span, per-slot, or per-token invocation, but the default systems interpretation is **triggered coarse-grained invocation with cached evidence**, not mandatory per-token database access.

Proposition 6.0.1 (Neutral non-invocation). If $\chi_t = 0$ and no cached episode is valid, then $\ell_t^{\mathrm{final}} = \ell_t^{\mathrm{model}}$.

Proof. In this case $\omega_t^{\mathrm{db}}(y) = 0$ for all y. Logit fusion gives $\ell_t^{\mathrm{model}}(y) + 0 = \ell_t^{\mathrm{model}}(y)$. □

Proposition 6.0.2 (Cache reuse preserves semantics). If a cached evidence episode is valid at token t, then using ω_e at t is semantically equivalent to re-executing the same deterministic program under the same relevant database and context state.

Proof. The validity predicate ν_e is defined to hold only when the state variables on which the episode depends have not changed in a way that would alter its deterministic result. Therefore re-execution would return the same evidence map ω_e. □

6.1 Model Forward Pass

The neural model computes

$$\ell_t^{\mathrm{model}} = F_\theta(x_{\le t}, s_t), \tag{79}$$

where s_t is any internal neural state.

6.2 Program Compilation

When $\chi_t = 1$, the compiler maps model logits and the current VM state to a typed chain of instructions:

$$P_t = \kappa(\ell_t^{\mathrm{model}}, s_t^{\mathrm{vm}}, \Gamma). \tag{80}$$

When $\chi_t = 0$, P_t is the distinguished `NOOP HALT` program.

6.3 Database Execution

The database VM executes the program:

$$s_{t+1}^{\mathrm{vm}} = \mathrm{Eval}_{\mathcal{M}_t}(P_t, s_t^{\mathrm{vm}}). \tag{81}$$

The halted state contains output accumulator

$$\omega_t : Y \to \mathbb{R}. \tag{82}$$

6.4 Logit Fusion

The final output logits are

$$\ell_t^{\mathrm{final}}(y) = \ell_t^{\mathrm{model}}(y) + \omega_t^{\mathrm{db}}(y), \tag{83}$$

where $\omega_t^{\mathrm{db}}(y) = 0$ whenever the VM has no evidence about y. Thus absence of a database contribution is PoE-neutral. If the VM accumulator and model output domains differ, a typed `ACCUM` projection converts database tuples into the model output domain before fusion.

6.5 Decoding

Deterministic decoding uses

$$y_t = \mathrm{argmax}_<(\ell_t^{\mathrm{final}}). \tag{84}$$

Probabilistic decoding, if desired, uses

$$y_t \sim \mathrm{softmax}(\ell_t^{\mathrm{final}}). \tag{85}$$

The VM remains deterministic in either case.

---

7. Fundamental Theorems

7.1 Closure

Theorem 7.1.1 (Closure of logit-valued query execution). Let R and S be logit-valued relations over finite domains. The instructions `UNION`, `INTERSECT`, `DIFFERENCE`, `FILTER`, `JOIN`, `PROJECT`, `TRAVERSE`, and `SCORE` produce logit-valued relations over finite domains.

Proof. Each instruction is defined by a finite expression using the semiring operations $\oplus$ and $\otimes$, deterministic predicates, and finite aggregation over finite domains. Since K is closed under $\oplus$ and $\otimes$, each resulting tuple receives a value in K. Therefore the output is a logit-valued relation. □

7.2 Support Preservation

Theorem 7.2.1 (Boolean algebra as support projection). Let B, C be classical relations and let η be the Boolean embedding. Then:

$$\mathrm{supp}(\eta(B) \cup_K \eta(C)) = B \cup C, \tag{86}$$

$$\mathrm{supp}(\eta(B) \cap_K \eta(C)) = B \cap C, \tag{87}$$

and

$$\mathrm{supp}(\eta(B) \setminus_K \eta(C)) = B \setminus C. \tag{88}$$

Proof. For union, a tuple has finite weight under $\eta(B) \cup_K \eta(C)$ exactly when it has finite weight in at least one of $\eta(B)$ or $\eta(C)$, which holds exactly when it belongs to $B \cup C$.

For intersection, a tuple has finite weight under $\eta(B) \cap_K \eta(C)$ exactly when both inputs have finite weight, which holds exactly when it belongs to $B \cap C$.

For difference, the definition explicitly preserves tuples in B whose support is absent from C. Therefore the support is $B \setminus C$. □

Corollary 7.2.2 (Classical query algebra recovery). Classical relational and posting-list query algebra is recovered from LVM execution by applying support projection to Boolean-embedded relations.

7.3 Determinism

Theorem 7.3.1 (Determinism of VM execution). For a fixed database state $\mathcal{M}$, fixed program P, and fixed initial VM state s_0, LVM execution produces at most one halted state.

Proof. Each instruction denotes a deterministic partial function on states. The program counter selects exactly one instruction at each step. Therefore the next state is uniquely determined whenever execution is defined. By induction on the number of executed steps, the entire execution trace is unique. If the program halts, the halted state is unique. □

7.4 Compositionality

Theorem 7.4.1 (Program composition). Let P and Q be two terminating LVM programs. Let $P; Q$ denote sequential composition. Then

$$\mathrm{Eval}_{\mathcal{M}}(P; Q, s) = \mathrm{Eval}_{\mathcal{M}'}(Q, \mathrm{Eval}_{\mathcal{M}}(P, s)), \tag{89}$$

where $\mathcal{M}'$ is the database state produced by executing P.

Proof. Sequential composition executes all instructions of P followed by all instructions of Q. Since small-step execution is deterministic, the state at the boundary between P and Q is exactly $\mathrm{Eval}_{\mathcal{M}}(P, s)$. The remainder is the evaluation of Q from that boundary state. □

7.5 Logit Aggregation as Product Composition

Theorem 7.5.1 (Logit addition equals normalized product composition). Let $\ell_1, \ell_2 : Y \to \mathbb{R}$ be two logit vectors. Define

$$\ell_{\text{sum}}(y) = \ell_1(y) + \ell_2(y). \tag{90}$$

Then

$$\text{softmax}(\ell_{\text{sum}})(y) = \frac{\text{softmax}(\ell_1)(y)\,\text{softmax}(\ell_2)(y)}{\sum_{u \in Y} \text{softmax}(\ell_1)(u)\,\text{softmax}(\ell_2)(u)}. \tag{91}$$

Proof. Expanding the left side:

$$\text{softmax}(\ell_{\text{sum}})(y) = \frac{\exp(\ell_1(y) + \ell_2(y))}{\sum_{u \in Y} \exp(\ell_1(u) + \ell_2(u))}. \tag{92}$$

This equals

$$\frac{\exp(\ell_1(y))\exp(\ell_2(y))}{\sum_{u \in Y} \exp(\ell_1(u))\exp(\ell_2(u))}. \tag{93}$$

Multiplying numerator and denominator by the normalizing constants of $\text{softmax}(\ell_1)$ and $\text{softmax}(\ell_2)$ gives the stated formula. □

Corollary 7.5.2 (Database evidence as an expert). If an LVM program returns output accumulator ω, then

$$\text{softmax}(\ell_{\text{model}} + \omega) \tag{94}$$

is the normalized product of the model distribution and the database-induced distribution. The execution that produced ω is deterministic.

7.6 Semantic Glue Localization

The tuple-to-output projection cannot be assumed away. When X is an entity domain and Y is an open vocabulary, the map

$$\psi : X \times Y \to K \tag{95}$$

is a substantive semantic object. If an entity has a canonical name, ψ may be a deterministic lookup. If the output is open vocabulary, multilingual, context-sensitive, or alias-heavy, ψ must be lexicalized or learned.

The correct claim is therefore not that semantic glue disappears. The correct claim is that the LVM **localizes** semantic glue into typed VM components: the compiler and the `ACCUM` projection kernel.

Theorem 7.6.1 (Localization of the semantic integration layer). Suppose a model-database integration has the following form:

1. A model state is mapped to a typed query expression.
2. The query expression is evaluated by database algebra over indexes.
3. The result is mapped to output-space evidence.
4. The evidence is added to model logits.

If step 1 is expressible by the typed compiler, step 2 is expressible in the instruction algebra of Section 4, and step 3 is expressible by a typed projection kernel ψ, then the entire integration is representable as LVM execution followed by logit fusion.

Proof. Step 1 is represented by the compiler κ. Step 2 is represented by a finite sequence of LVM instructions because the query expression is assumed expressible in the instruction algebra. Step 3 is represented by `ACCUM` with typed projection kernel ψ. Step 4 is the logit fusion rule of Section 6.4. Therefore the integration is an LVM execution plus deterministic logit addition. The semantic boundary is not external application code; it is one of the typed operators of the VM. □

Corollary 7.6.2 (Glue as a verifiable operator). Any remaining semantic conversion must appear either in the compiler grammar or in ψ. Both are typed, trainable, testable, and auditable. There is no untyped model-database conversion layer outside the semantics.

Remark 7.6.3 (Scope). The theorem does not claim that no code is executed. It claims that the semantic content of the integration is captured by typed VM components. Physical indexes and kernels evaluate the program, and projection kernels map database tuples to output evidence, but neither defines an ad hoc model-database interface.

---

8. Memory Writes and Transactions

8.1 The Write Problem

A durable memory system cannot treat writing as overwriting a differentiable matrix cell. Database memory must support:

- typed records,
- consistency constraints,
- versioning,
- transactions,
- conflict resolution,
- indexing,
- deletion and correction,
- auditability.

Thus write semantics must be database semantics.

8.2 Typed Memory Delta

A memory write is represented by a typed delta

$$\Delta = (\mathrm{op}, \mathrm{type}, \mathrm{id}, \mathrm{payload}, \mathrm{evidence}, \mathrm{time}), \tag{96}$$

where:

- op is `insert`, `update`, `merge`, or `delete`.
- type is a schema type.
- id is an entity or record identifier.
- $\mathrm{payload}$ is the proposed value.
- $\mathrm{evidence}$ is a logit-valued confidence or provenance field.
- time is a validity or transaction timestamp.

8.3 Write Instruction

The `WRITE` instruction appends a delta to a transaction buffer:

$$\rho'(b) = \rho(b) \mid\mid \Delta, \tag{97}$$

where $\mid\mid$ denotes append.

The durable database state is not changed by `WRITE` alone.

8.4 Commit Instruction

The `COMMIT` instruction applies a deterministic transaction function

$$\mathrm{commit} : \mathcal{M} \times \mathrm{Buffer} \to \mathcal{M}. \tag{98}$$

The new database state is

$$\mathcal{M}' = \mathrm{commit}(\mathcal{M}, \rho(b)). \tag{99}$$

The transaction function enforces schema constraints, index maintenance, and conflict policy.

Theorem 8.4.1 (Deterministic durability). If the commit function is deterministic, then for a fixed initial database state and fixed transaction buffer, `COMMIT` produces a unique durable database state.

Proof. Immediate from determinism of commit. □

8.5 Memory as State Transition

The complete write path is therefore

$$(\ell_t, \mathcal{M}_t) \to_\kappa P_t \to_{\mathrm{Eval}} (\omega_t, \mathcal{M}_{t+1}). \tag{100}$$

The model may influence the write through logits, but the database enforces the write through typed deterministic transition semantics.

---

9. Indexes as Physical Evaluators

9.1 Semantic and Physical Layers

The LVM distinguishes semantic instructions from physical plans.

A semantic instruction is a mathematical operator such as

$$\mathrm{Filter}_\varphi, \quad \mathrm{Project}_\pi, \quad \mathrm{Traverse}_r, \quad \mathrm{Score}_e. \tag{101}$$

A physical plan is an implementation using indexes, caches, postings, vector graphs, joins, sorted lists, block maxima, or distributed execution.

The semantic operator is independent of the physical plan.

9.2 Text Indexes

A text index evaluates instructions such as `SEEK_TERM` and `SCORE_TEXT`. For term t, the Boolean candidate relation is

$$L_t(d) = \begin{cases} 0, & t \in d, \\ -\infty, & t \notin d. \end{cases} \tag{102}$$

A scoring instruction adds deterministic text evidence:

$$\mathrm{ScoreText}(L_t)(d) = L_t(d) + e_{\mathrm{text}}(d, t). \tag{103}$$

9.3 Vector Indexes

A vector index evaluates `SEEK_VECTOR` and `SCORE_VECTOR`. Let q be a query vector, and let C_q be the candidate set returned by a deterministic ANN search configuration. The candidate relation is

$$L_q(d) = \begin{cases} 0, & d \in C_q, \\ -\infty, & d \notin C_q. \end{cases} \tag{104}$$

The scoring instruction adds vector evidence:

$$\mathrm{ScoreVector}(L_q)(d) = L_q(d) + e_{\mathrm{vec}}(d, q, \mathcal{T}). \tag{105}$$

The function e_{vec} may depend on distance, local density, cell population, graph trajectory, or other deterministic index statistics.

9.4 Graph Indexes

A graph index evaluates traversal and pattern instructions. If A_r is the adjacency relation for edge label r, then traversal is semiring multiplication:

$$F' = FA_r. \tag{106}$$

Pattern matching is represented by repeated join, filter, and projection over edge and vertex relations.

9.5 Exact Pruning

If an instruction or subprogram has a known upper bound on possible future logit contribution, exact pruning is possible.

Let the current best output margin be

$$m = \ell(y_1) - \ell(y_2), \tag{107}$$

where y_1 and y_2 are the current top two outputs. Let U be an upper bound on all remaining possible contribution to any competitor. If

$$m > U, \tag{108}$$

then no remaining computation can change the top-1 output.

Theorem 9.5.1 (Top-1 pruning safety). Under the bound condition $m > U$, skipping the remaining bounded subprogram preserves the top-1 output.

Proof. The strongest possible future increase of any competitor is at most U. Since the current top-1 margin exceeds U, every competitor remains below the current top-1 even after maximal allowed future contribution. Therefore top-1 cannot change. □

---

10. Worked Example

Consider the user input:

```
Who founded the company that acquired Instagram?
```

A neural model produces logits over operations and symbols. The compiler emits the following program:

```
LOAD_SYMBOL "Instagram" -> s0
SEEK_TERM s0 -> r0
SCORE_TEXT r0 -> r1
PROJECT r1 entity.company -> r2
TRAVERSE r2 acquired_by 1 -> r3
PROJECT r3 entity.founder -> r4
ACCUM r4 entity_to_token -> out
HALT
```

The execution is deterministic:

1. `SEEK_TERM` finds records associated with "Instagram".
2. `SCORE_TEXT` adds text evidence.
3. `PROJECT` maps records to company entities.
4. `TRAVERSE` follows the acquisition edge.
5. `PROJECT` maps the resulting company to founder entities.
6. `ACCUM` maps founder entities to token logits.

The database returns an output accumulator

$$\omega : V \to K, \tag{109}$$

where V is the token vocabulary. The final decoder logits are

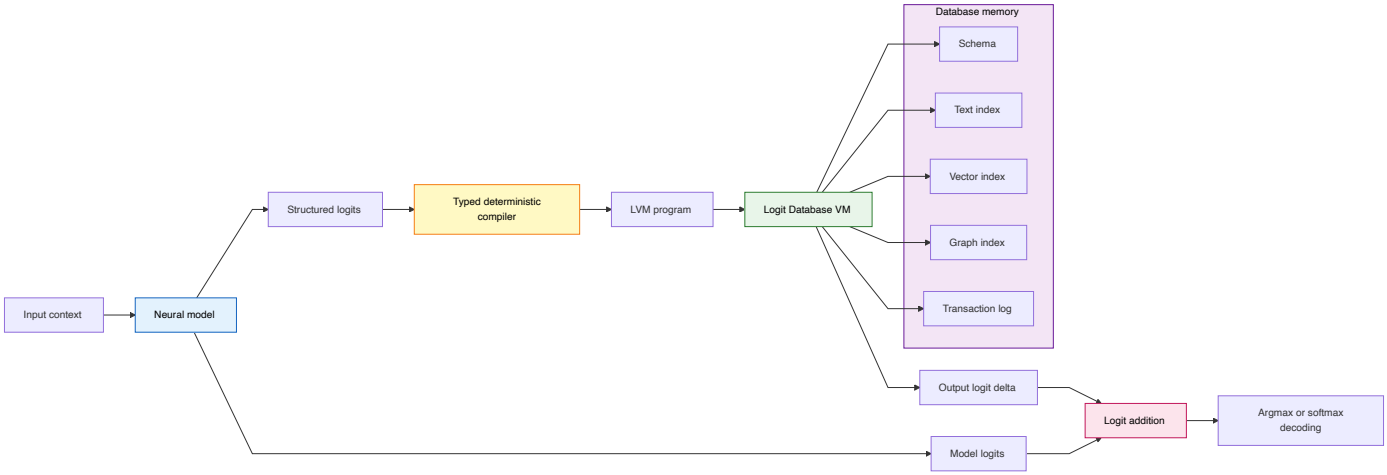
$$\ell_{\text{final}} = \ell_{\text{model}} + \omega. \tag{110}$$

No retrieved paragraph needs to be serialized back into the prompt. The database has executed a typed program and returned logit evidence.

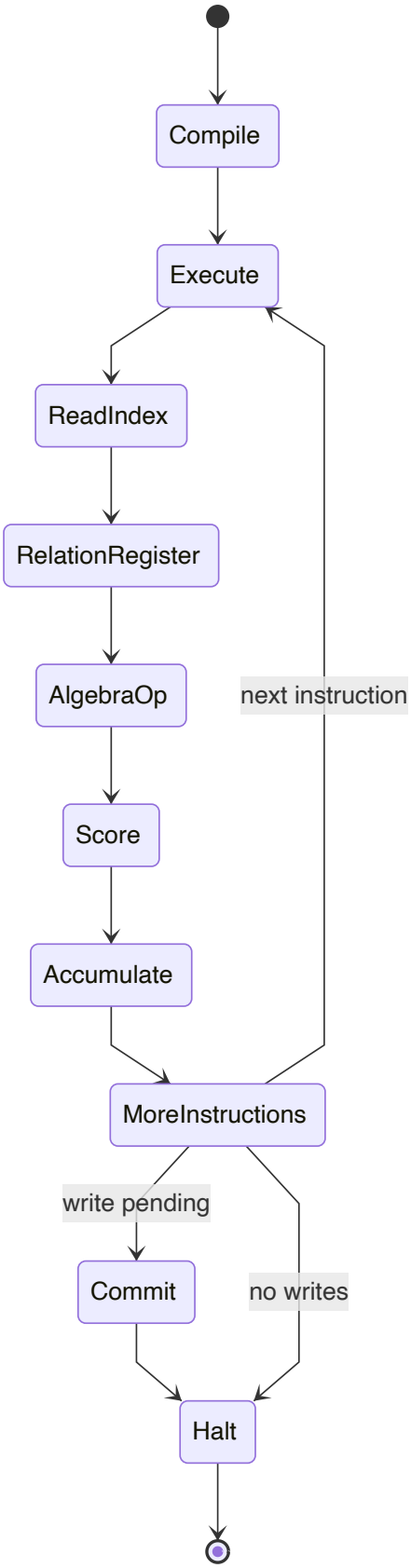
---

11. Diagrams

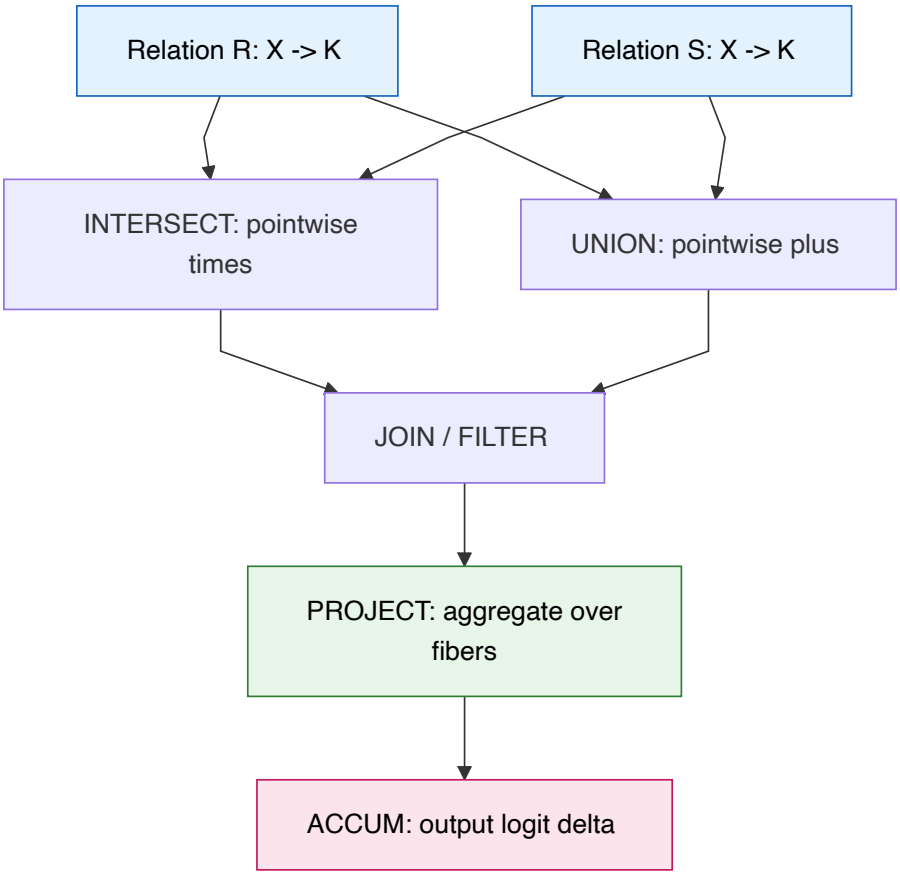
11.1 LVM Inference Architecture



11.2 VM State Transition



11.3 Semiring Query Execution



12. Relationship to Existing Memory Models

12.1 Differentiable Memory

Differentiable memory architectures represent memory as tensors and implement read/write through differentiable addressing. This makes memory trainable end-to-end, but it also constrains memory to live inside the neural computation graph.

The LVM takes the opposite position. Memory is not a tensor. It is durable typed state. Its operations are query execution, indexing, traversal, projection, scoring, and transaction commit. The neural model communicates with this state through logits and typed instructions.

12.2 Tool Use

Tool use treats the database as an external callable function. The model emits a tool call; the environment returns text or structured data.

The LVM treats the database as an execution substrate. A tool call is an opaque boundary; an LVM program is a typed algebraic object with deterministic semantics.

12.3 Retrieval-Augmented Generation

Retrieval-augmented generation inserts retrieved text into the model context. This is a practical strategy, but it makes integration depend on serialization, chunking, prompt layout, and context-window allocation.

The LVM replaces retrieved context with output-space evidence:

$$\text{retrieved records} \to \Delta\ell. \tag{111}$$

The model need not reread its own database output as text. It receives the database contribution directly in logit space.

13. Training the Neural Compiler and the LVM

The deterministic execution semantics of the LVM does not make training impossible. It changes what must be learned. The database evaluator does not need to be differentiable, and the compiler is not a hand-coded semantic bridge or a deferred external component. The model must learn a typed neural compiler whose selected programs produce output logits that improve the target.

The training problem is therefore not:

$$\text{Differentiate through database execution.} \tag{112}$$

It is:

$$\text{Learn a typed compiler whose deterministic programs yield correct output logits.} \tag{113}$$

Compiler learning is therefore a constructive part of the theory: grammar validity, stable deterministic decoding, supervised traces, self-supervised trace generation, inverse execution, pruned beam search, and annealed soft compilation together specify how the compiler is obtained.

This section gives a training theory for both the compiler and the LVM.

13.1 Bounded Typed Program Space

Let Γ be a typing context and let T be a maximum program length. Let

$$\mathcal{P}_{\Gamma,T} \tag{114}$$

be the finite set of all well-typed LVM programs of length at most T under context Γ.

A neural model and compiler together produce two objects for input x:

1. base output logits

$$b_\theta(y \mid x), \tag{115}$$

2. program scores induced by typed derivation scores

$$S_\theta(P \mid x), \qquad P \in \mathcal{P}_{\Gamma,T}. \tag{116}$$

When programs are produced through grammar derivations D and deterministic lowering $P = \text{lower}_<(D)$, the program score is induced by the compiler score:

$$S_\theta(P \mid x) = \max_{D:\text{lower}_<(D)=P} S_\theta(D \mid x). \tag{117}$$

The score S_θ may be produced autoregressively by the compiler:

$$S_\theta(P \mid x) = \sum_{i=0}^{|P|-1} g_\theta(a_i \mid x, q_i, a_{<i}). \tag{118}$$

For a fixed database state $\mathcal{M}$, each program has deterministic execution result

$$\omega_P = \Omega_{\mathcal{M}}(P, x), \tag{119}$$

where

$$\omega_P : Y \to K \tag{120}$$

is the output logit delta returned by the LVM.

The program execution $P \mapsto \omega_P$ is not required to be differentiable.

13.2 Conditional Output Distribution Given a Program

Given a program P, the final output distribution is

$$p_\theta(y \mid x, P) = \text{softmax}(b_\theta(\cdot \mid x) + \omega_P)(y). \tag{121}$$

The database result ω_P is a deterministic feature of P, x, and $\mathcal{M}$. The trainable quantities are the neural base logits b_θ, the compiler program scores S_θ, and optionally the projection parameters inside `ACCUM`.

13.3 Latent Execution Likelihood

When the correct program is not observed, treat it as a latent variable. Define the neural program prior

$$p_\theta(P \mid x) = \frac{\exp(S_\theta(P \mid x))}{\sum_{Q \in \mathcal{P}_{\Gamma,T}} \exp(S_\theta(Q \mid x))}. \tag{122}$$

The marginal likelihood of the target output y is

$$p_\theta(y \mid x) = \sum_{P \in \mathcal{P}_{\Gamma,T}} p_\theta(P \mid x)\, p_\theta(y \mid x, P). \tag{123}$$

The training loss is

$$\mathcal{L}_{\text{exec}}(\theta) = -\log p_\theta(y \mid x). \tag{124}$$

This objective trains the compiler to assign probability mass to programs whose deterministic execution raises the correct output.

13.4 Program Posterior and Credit Assignment

Define the posterior responsibility of program P after observing target y:

$$q_\theta(P \mid x, y) = \frac{p_\theta(P \mid x)\, p_\theta(y \mid x, P)}{\sum_{Q \in \mathcal{P}_{\Gamma,T}} p_\theta(Q \mid x)\, p_\theta(y \mid x, Q)}. \tag{125}$$

Programs that make the target more likely receive larger responsibility. This posterior gives credit assignment over discrete programs without differentiating through the database.

If the program score decomposes over instructions,

$$S_\theta(P \mid x) = \sum_{i=0}^{|P|-1} s_\theta(a_i \mid x, a_{<i}), \tag{126}$$

then the program posterior distributes credit to instruction logits through the standard gradient of S_θ. The database execution remains a deterministic evaluator.

13.5 The No-Database-Gradient Theorem

Theorem 13.5.1 (Trainability without differentiating through execution). For finite $\mathcal{P}_{\Gamma,T}$, the latent execution loss $\mathcal{L}_{\text{exec}}$ is differentiable with respect to θ wherever b_θ and S_θ are differentiable, even if the execution map $P \mapsto \omega_P$ is non-differentiable.

Proof. The loss is a finite composition of differentiable functions of b_θ and S_θ: exponentials, finite sums, logarithms, and softmax terms. For each fixed program P, ω_P is a constant with respect to θ unless the `ACCUM` projection is parameterized. Therefore no derivative of the execution map is needed. If `ACCUM` has parameters, gradients flow only through that projection map, not through seek, join, traversal, filter, or commit. □

Remark 13.5.2 (Why this is not reinforcement learning by necessity). The objective can be optimized by enumerating or beam-selecting a finite candidate set and computing exact responsibilities over that set. No stochastic policy-gradient estimator is required unless the candidate space is sampled rather than enumerated.

13.6 Supervised Compiler Learnability

When gold traces are available, the compiler is trained directly.

For a gold program

$$P^* = (a_0^*, \ldots, a_{n-1}^*), \tag{127}$$

define the trace loss

$$\mathcal{L}_{\text{trace}}(\theta) = -\sum_{i=0}^{n-1} \log p_\theta(a_i^* \mid x, q_i, a_{<i}^*). \tag{128}$$

Theorem 13.6.1 (Separable compiler convergence). Suppose a training set has gold programs P_i^* and the compiler class can realize a positive margin $m > 0$:

$$S_{\theta^*}(P_i^* \mid x_i) \geq S_{\theta^*}(P \mid x_i) + m \tag{129}$$

for every i and every $P \neq P_i^*$ in $\mathcal{P}_{\Gamma,T}$. Then there exists a sequence of scaled parameters whose trace loss tends to zero and whose deterministic compiler emits P_i^* for every training input x_i.

Proof. Scale the separating scores by $c > 0$. The probability assigned to P_i^* is

$$\frac{\exp(cS_{\theta^*}(P_i^* \mid x_i))}{\sum_P \exp(cS_{\theta^*}(P \mid x_i))}. \tag{130}$$

All non-gold programs are lower by at least cm, so their total probability tends to zero as $c \to \infty$. Hence the trace loss tends to zero. The stable argmax compiler selects the unique highest-scoring gold program. □

This theorem solves the supervised compiler case: if the desired programs are representable and separable, the deterministic compiler is learned by ordinary cross-entropy.

13.7 Latent Compiler Learnability from Answers

Gold traces are useful but not required. Suppose only (x, y) is observed. Define the execution quality of a program by

$$r(P; x, y) = p_\theta(y \mid x, P). \tag{131}$$

Theorem 13.7.1 (Execution-optimal compiler target). For fixed base logits and fixed execution deltas, the latent execution likelihood

$$\sum_P p_\theta(P \mid x) r(P; x, y) \tag{132}$$

is maximized by assigning all program mass to any program $P^\dagger$ with maximal execution quality

$$P^\dagger \in \arg\max_P r(P; x, y). \tag{133}$$

If the maximizer is unique, the optimal deterministic compiler emits $P^\dagger$.

Proof. The likelihood is a convex combination of the values $r(P; x, y)$. A convex combination is maximized by placing all mass on an element with maximal value. If the maximizer is unique, all optimal mass lies on that program, and stable argmax emits it. □

Remark 13.7.2 (Learning without traces). This theorem gives the answer-only training principle: search for programs whose execution makes the observed answer likely, then train the compiler to prefer those programs. The database supplies deterministic evidence; the target supplies credit assignment.

13.8 Finite-State Typed Beam Training

The full program space may be large, but the compiler is finite-state and typed. Training uses a deterministic beam over grammar states.

Let

$$B_\theta(x) \subset \mathcal{P}_{\Gamma,T} \tag{134}$$

be the deterministic top-B candidate set produced by the typed beam compiler. Define the beam likelihood

$$p_\theta^B(y \mid x) = \sum_{P \in B_\theta(x)} p_\theta^B(P \mid x)\, p_\theta(y \mid x, P), \tag{135}$$

where

$$p_\theta^B(P \mid x) = \frac{\exp(S_\theta(P \mid x))}{\sum_{Q \in B_\theta(x)} \exp(S_\theta(Q \mid x))}. \tag{136}$$

Proposition 13.8.1 (Typed beam complexity). Let $b_{\max}$ be the maximum number of valid next fragments from any grammar state. A typed beam compiler with width B and maximum length T performs

$$O(TBb_{\max}) \tag{137}$$

compiler scoring steps, plus deterministic execution of at most B candidate programs.

Proof. At each of T steps, at most B prefixes are active. Each prefix expands to at most $b_{\max}$ valid next fragments because invalid fragments are masked by the grammar. The stated bound follows. □

Remark 13.8.2 (Why search is part of the solution). Search is not an unsolved compiler problem; it is the finite-state execution strategy of the compiler. Type masks, register constraints, cost bounds, memoized partial execution, and admissible upper bounds are compiler primitives.

13.9 Pruned Compiler Search

Let $U(P_{<i})$ be an admissible upper bound on the best possible final score of any completion of prefix $P_{<i}$:

$$S_\theta(P \mid x) \leq U(P_{<i}) \tag{138}$$

for every completion P of the prefix. If the current beam threshold is β_B, any prefix satisfying

$$U(P_{<i}) < \beta_B \tag{139}$$

can be discarded without changing the final top-B beam.

Theorem 13.9.1 (Safe compiler pruning). Admissible prefix pruning preserves the deterministic top-B program set.

Proof. If $U(P_{<i}) < \beta_B$, then no completion of the prefix can reach the current beam threshold. Therefore no completion can enter the final top-B set. Discarding the prefix cannot remove a top-B program. □

This is the compiler analogue of exact database pruning: upper bounds make discrete program search tractable without changing semantics.

13.10 Self-Supervised Trace Generation as Cold-Start Solution

Latent execution likelihood alone is a weak cold-start signal. A randomly initialized compiler is unlikely to sample a program whose execution supports the target. The practical training solution is therefore to generate traces from the database itself before relying on latent answer-only credit assignment.

A trace generator samples a well-typed program P, executes it on $\mathcal{M}$, and produces a training triple:

$$(x_P, y_P, P). \tag{140}$$

Examples include:

- sample an entity and ask for one of its fields,
- sample a relation and ask for a joined attribute,
- sample a graph path and ask for its endpoint,
- sample a text term and ask for a document or entity it identifies,
- sample an alias and ask for the canonical entity,
- sample a transaction delta and ask whether it satisfies schema constraints,
- sample a program and ask for the output symbol whose logit should be increased by `ACCUM`.

This gives direct supervision for the neural compiler without requiring hand-labeled traces.

Definition 13.10.1 (Trace-generating grammar). A trace-generating grammar is a probability distribution

$$G_{\text{trace}}(P \mid \Gamma, \mathcal{M}) \tag{141}$$

over well-typed programs, together with deterministic input and answer renderers

$$\mathsf{Render}_x(P, \mathcal{M}), \qquad \mathsf{Render}_y(P, \mathcal{M}). \tag{143}$$

It produces

$$(x_P, y_P, P) = (\mathsf{Render}_x(P, \mathcal{M}), \mathsf{Render}_y(P, \mathcal{M}), P). \tag{144}$$

Proposition 13.10.2 (Compiler cold-start coverage). Suppose every production in the typed compiler grammar appears with non-zero probability under G_{trace}, and every schema type is sampled with non-zero probability. Then every bounded well-typed program schema appears in the self-supervised trace stream with non-zero probability.

Proof. A bounded program schema is a finite sequence of grammar productions and typed argument choices. Each has non-zero probability by assumption. The product of finitely many non-zero probabilities is non-zero. □

Training schedule. The natural curriculum has three phases:

1. **Trace pretraining.** Optimize mostly $\mathcal{L}_{\text{trace}}$ on generated traces so that the compiler learns the grammar, schema, register discipline, active-view protocol, and projection boundary.
2. **Mixed execution training.** Mix generated traces with inverse-execution pseudo-traces and latent execution likelihood.
3. **Latent fine-tuning.** Reduce trace weight and optimize task likelihood, cost, calibration, and evidence budget.

A practical implementation may make generated traces dominate the early curriculum and let latent execution dominate only after the compiler can reliably produce executable programs. The exact ratio is an engineering hyperparameter, but the structural point is fixed: self-supervised traces solve compiler cold start.

13.11 Inverse Execution for Weak Supervision

When only (x, y) is observed, the system can search for programs whose execution supports y. Define the execution reward

$$r(P; x, y) = \log \operatorname{softmax}(b_\theta(\cdot \mid x) + \omega_P)(y). \tag{145}$$

A bounded planner searches for high-reward programs:

$$P^+ \in \arg\max_{P \in B_\theta(x)} r(P; x, y). \tag{146}$$

The resulting pseudo-trace P^+ can be used in $\mathcal{L}_{\text{trace}}$, or the reward can be used directly in the latent execution likelihood. This is inverse execution: programs are discovered by the evidence they contribute to the observed target.

13.12 Annealed Soft Compiler

The deterministic compiler used at inference is the zero-temperature limit of a soft compiler used during training. For temperature $\tau > 0$, define

$$p_{\theta,\tau}(P \mid x) = \frac{\exp(S_\theta(P \mid x)/\tau)}{\sum_Q \exp(S_\theta(Q \mid x)/\tau)}. \tag{147}$$

The relaxed output distribution is

$$p_{\theta,\tau}(y \mid x) = \sum_P p_{\theta,\tau}(P \mid x)\, p_\theta(y \mid x, P). \tag{148}$$

Training begins with larger τ, allowing multiple plausible programs to receive credit, and gradually anneals τ toward zero.

Theorem 13.12.1 (Zero-temperature convergence to deterministic compilation). Suppose $S_\theta(P \mid x)$ has a unique maximizer

$$P^* = \arg\max_P S_\theta(P \mid x). \tag{149}$$

Then

$$\lim_{\tau \to 0} p_{\theta,\tau}(P \mid x) = \begin{cases} 1, & P = P^*, \\ 0, & P \neq P^*. \end{cases} \tag{150}$$

Consequently,

$$\lim_{\tau \to 0} p_{\theta,\tau}(y \mid x) = p_\theta(y \mid x, P^*). \tag{151}$$

Proof. This is the standard zero-temperature limit of the softmax distribution. The unique maximizer dominates the denominator exponentially as $\tau \to 0$. Substitution into the relaxed output distribution yields the deterministic program-conditioned output. □

Corollary 13.12.2 (Training-time relaxation preserves deterministic inference). Annealed training does not alter the deterministic VM semantics. It provides a differentiable learning objective whose zero-temperature limit is the inference-time compiler.

13.13 Learning the `ACCUM` Projection Boundary

The projection

$$\psi : X \times Y \to K \tag{152}$$

inside `ACCUM` is the explicit semantic boundary between database tuples and model outputs. It can be fixed, lexicalized, learned, or hybrid.

A fixed projection is appropriate when database tuples map exactly to output symbols, such as entity IDs, canonical names, schema values, enumerations, or foreign keys.

A lexicalized projection is appropriate when a tuple has a finite alias set:

$$\psi(x, y) = \lambda_{\text{alias}}(x, y). \tag{153}$$

A learned projection has parameters ϕ:

$$\psi_\phi(x, y). \tag{154}$$

Then

$$\chi_{R_P,\psi_\phi}(y) = \begin{cases} \bigoplus_{x \in C_y(R_P,\psi_\phi)} R_P(x) \otimes \psi_\phi(x, y), & C_y(R_P, \psi_\phi) \neq \emptyset, \\ 0, & C_y(R_P, \psi_\phi) = \emptyset, \end{cases} \tag{155}$$

and

$$\omega_P(y) = \omega_0(y) + \chi_{R_P,\psi_\phi}(y). \tag{156}$$

Gradients flow to ϕ through the final output loss. This learns how tuple evidence should affect output symbols while leaving database execution itself deterministic.

Verification constraints. Projection learning is constrained by:

- type signatures: ψ may only connect compatible tuple and output types;
- support constraints: tuples outside relation support do not contribute;
- evidence neutrality: absent output support contributes 0, not $-\infty$;
- evidence budget: ω cannot grow without calibration;
- alias tests: known canonical names and aliases form supervised anchors;
- counterfactual tests: unrelated entities should project neutral or negative evidence.

Thus ψ is not hidden glue. It is a named, typed, trainable, and testable operator.

13.14 Evidence Budget Regularization

Because database evidence is added directly to logits, it should be calibrated. Introduce an evidence budget penalty:

$$\mathcal{L}_{\text{budget}} = \lambda_\omega \, \|\omega_P\|_2^2 + \lambda_\Delta \, \max(0, \|\omega_P\|_\infty - c). \tag{157}$$

The first term discourages diffuse over-amplification. The second enforces a hard maximum evidence magnitude c up to a hinge. This keeps the database from overwhelming the base model unless execution evidence is strong.

13.15 Joint Training Objective

The full training objective is

$$\mathcal{L}(\theta, \phi) = \mathcal{L}_{\text{exec}} + \lambda_{\text{trace}}\mathcal{L}_{\text{trace}} + \lambda_{\text{schema}}\mathcal{L}_{\text{schema}} + \lambda_{\text{ground}}\mathcal{L}_{\text{ground}} + \lambda_{\text{inv}}\mathcal{L}_{\text{inverse}} + \lambda_{\text{base}}\mathcal{L}_{\text{base}} + \lambda_C\mathcal{L}_{\text{cost}} + \mathcal{L}_{\text{budget}}. \tag{158}$$

Here $\mathcal{L}_{\text{schema}}$ is the schema-generated trace curriculum of Section 13.10, and $\mathcal{L}_{\text{ground}}$ trains canonical, alias-based, tokenizer-based, and learned `ACCUM` projections. $\mathcal{L}_{\text{base}}$ is ordinary next-token or task loss without database execution. This prevents the model from becoming dependent on database programs when the answer is already available from the neural state. The inverse-execution term trains the compiler from pseudo-traces discovered by execution reward, and the cost term makes short, cheap, well-typed programs preferable

when two programs explain the target equally well.

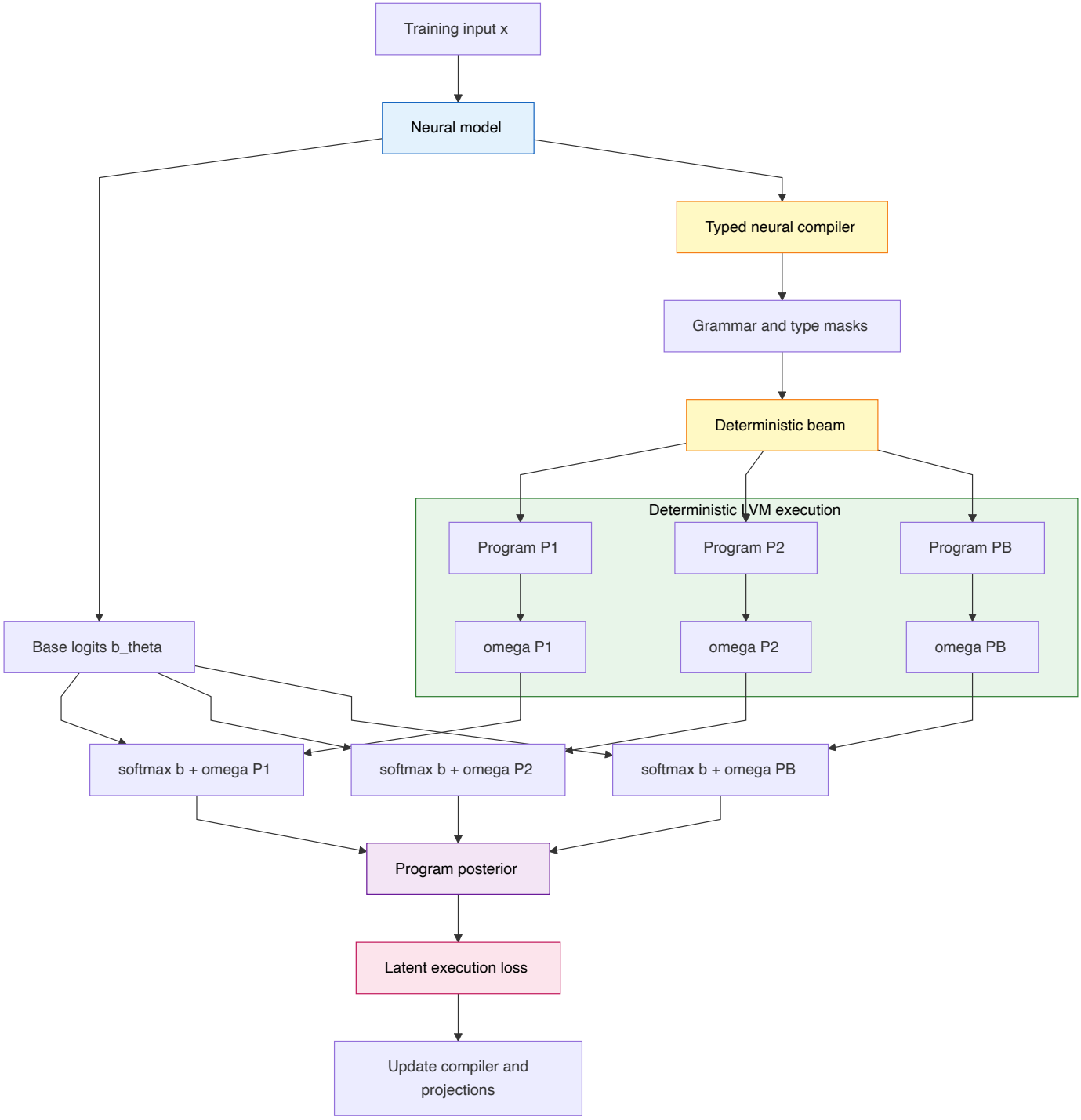
13.16 Training Algorithm

```
Input: training pair (x, y), database state M, beam width B
Output: parameter update for compiler theta and projection phi

0. Bootstrap phase:
      generate schema-driven self-supervised traces (x_P, y_P, P)
      train compiler on trace loss until valid-program recall is high
      train ACCUM projection psi_phi on canonical, alias, and typed output mappings

1. Neural model computes base logits b_theta(. | x)
2. Trigger policy decides whether LVM invocation is needed
3. If no trigger fires:
      omega(y) = 0 for all y
      train ordinary base loss if applicable
4. If a trigger fires:
      typed neural compiler begins an interactive chain-of-instruction
5. At each compiler step:
      materialize finite candidate sets from grammar, schema, registers, and indexes
      score only the materialized candidates
      mask invalid fragments by type
      optionally execute partial instruction to materialize the next candidate set
6. Deterministic typed beam compiler produces candidate programs B_theta(x)
7. For each program P in B_theta(x):
      execute P deterministically on M
      obtain output logit delta omega_P with zero evidence outside ACCUM support
      compute target likelihood softmax(b_theta + omega_P)[y]
8. Compute latent execution posterior over candidate programs
9. Minimize:
      execution loss
      + trace loss
      + inverse-execution pseudo-trace loss
      + base-model loss
      + evidence-budget penalty
      + cost penalty
10. Anneal compiler temperature toward zero during training
11. At inference, use triggered invocation and deterministic stable-argmax compilation
```

13.17 Training Diagram



13.18 Interpretation

The training solution has a clean separation of roles:

- The database is a deterministic evaluator.
- The compiler is a finite-state typed neural scorer.
- The verifier masks invalid fragments before execution.
- Program selection is discrete at inference.
- Training uses latent deterministic execution to assign credit to programs.
- No instruction requires differentiability of indexes, joins, graph traversal, filters, or transactions.

Thus the compiler is not a remaining limitation. It is the trainable front end of the LVM, with supervised, self-supervised, and latent-execution learning paths.

---

14. Practical Engineering Considerations

14.1 Physical Plan Costs

The semantic VM abstracts physical execution. Real database engines must choose efficient plans, maintain indexes, distribute computation, and respect latency budgets. This is an optimizer problem, not a semantic gap. Cost-aware compilation in Section 5.7 and pruned compiler search in Section 13.9 provide the formal interface.

14.2 Concurrent Neural Agents

If multiple neural agents write to the same database state, concurrency control becomes central. The transaction semantics in Section 8 provide the formal hook. Conflict policy can be deterministic, versioned, or application-specific while preserving VM semantics.

14.3 Compiler Runtime Budgets

The typed compiler controls runtime through trigger policies, finite-state grammar masks, bounded program length, deterministic beam width, candidate materialization, admissible pruning, memoized partial execution, and cost-aware scoring. Runtime is therefore an explicit compiler budget, not a semantic limitation.

14.4 Calibration of the Projection Boundary

The `ACCUM` projection determines how database tuples become output logits. It is the main semantic boundary in the system. It can be trained by the same latent execution objective, anchored by self-supervised traces, constrained by evidence-budget regularization, and monitored by calibration metrics. Calibration is therefore part of the training objective rather than an external post-processing step.

14.5 Transactional Memory Writes

Writes can be trained with the same framework by treating proposed deltas as program outputs. A write is correct when the committed database state improves future likelihood, satisfies schema constraints, or matches a supervised transaction trace.

14.6 Large Entity Domains

Large entity and document spaces are handled by candidate materialization and pointer logits. The physical system must provide candidate generators, top-k scorers, relation-register summaries, and handle-based pointer selection. A flat entity logit vector is not part of the implementation contract.

14.7 Invocation Frequency and Caching

Runtime is governed by trigger frequency, episode cost, cache validity, and program length. The LVM may be invoked once per query, once per paragraph, once per schema slot, or only when uncertainty crosses a threshold. This is a scheduling decision over the same deterministic semantics, not a change to the model.

14.8 Grounding Auditability

Because ψ is first-class, grounding errors can be inspected directly: which tuple contributed to which output symbol, through which alias, tokenizer path, or learned projection, and with what evidence magnitude. This turns a hidden integration problem into an auditable VM trace.

---

15. Extensions

15.1 Learned Physical Optimizers

Use neural predictions to choose physical database plans while preserving deterministic semantic execution. This is a physical-plan optimizer for the database backend, distinct from the neural typed compiler of Section 5: the former chooses how to execute an already-selected program efficiently, while the latter chooses which well-typed program should be executed.

15.2 Multi-VM Composition

Compose multiple LVMs over different database states: personal memory, public knowledge, graph memory, temporal logs, and code repositories.

15.3 Transactional Neural Memory

Develop a full theory of neural memory writes as typed transactions, including merge policies, correction, forgetting, and provenance.

15.4 Hardware Analogy

Treat the LVM instruction set as an ISA for neural-database systems. Neural compilation lowers logits into this ISA; database indexes act as physical execution units; physical query optimizers select efficient plans for the already-typed instructions.

---

16. Conclusion

This paper introduced Logit Virtual Machines: deterministic database execution over neural natural parameters. The key move is to treat logits as executable state rather than merely as final decoding scores. A neural model emits structured logits; a learned typed compiler scores grammar derivations; deterministic lowering converts the selected derivation to VM instructions; a database VM executes those instructions over indexed durable memory; and the result is returned as output-space logit evidence.

The framework unifies Boolean query algebra, ranked retrieval, vector search, graph traversal, transactional memory writes, typed `ACCUM` projection, and trainable neural-to-database compilation under one semantics: logit-valued relations over a log semiring plus zero-neutral output evidence. Classical database behavior is recovered by support projection, while probabilistic interpretation is recovered by softmax after deterministic logit fusion.

The training problem does not require differentiating through the database, nor does it require assuming an external compiler. LVM training treats derivations and programs as latent structured variables, but compiler cold start is handled by schema-driven self-supervised traces before latent execution fine-tuning. The database executes candidate programs deterministically; the compiler learns, through trace supervision, candidate materialization, inverse execution, latent execution likelihood, and annealed compilation, which typed derivations produce output evidence that improves the target. In the zero-temperature limit, the training-time relaxation converges to the deterministic compiler used at inference.

A database in this framework is not a passive retrieval tool and not a differentiable memory matrix. It is a deterministic virtual machine whose state is durable, typed, indexed, transactional, and trainable through a verified neural compiler. The unavoidable projection from database tuples to output symbols is made explicit as a typed, trainable, calibrated operator. Neural-database integration becomes an execution problem over natural parameters, with its semantic boundary formalized rather than hidden in prompt-engineering code.

References

Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

Broder, A. Z., Carmel, D., Herscovici, M., Soffer, A., & Zien, J. (2003). Efficient query evaluation using a two-level retrieval process. *Proceedings of CIKM*.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 377–387.

Green, T. J., Karvounarakis, G., & Tannen, V. (2007). Provenance semirings. *Proceedings of PODS*.

Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8), 1771–1800.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2006). *Introduction to Automata Theory, Languages, and Computation*. Pearson.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*.

Weston, J., Chopra, S., & Bordes, A. (2015). Memory networks. *International Conference on Learning Representations*.

Zaremba, W., & Sutskever, I. (2015). Reinforcement learning neural Turing machines. *arXiv preprint*.